Improving Clang Error Recovery for Interactive Workflows

Name: Sahil Patidar
Email: sahilpatidar60@gmail.com
Github: https://github.com/SahilPatidar

Project Description

This project aims to improve error recovery in Clang, especially for interactive and incremental environments like clang-repl. The goal is to make the compiler handle incomplete or invalid C++ code more gracefully, so users can experiment, learn, and prototype without frequent interruptions. The work will focus on complex C++ features such as templates, argument-dependent lookup (ADL), and name conflicts. It will also improve stability, ensuring the compiler stays responsive and avoids crashes during interactive use.

Problem Statement

Clang's error recovery is limited in interactive use. Invalid or partial C++ code can leave the compiler in a broken state, especially with templates, ADL, or conflicting declarations. This reduces the reliability of tools like clang-repl. Recovery paths are also not well-tested, and some cases can cause crashes. The key challenge is to improve recovery without hiding real errors or breaking the compiler's internal state.

Expected Outcomes

- Expanded test coverage for diverse error recovery scenarios
- Identification and fixes for existing recovery-related bugs

- Improved recovery handling for: Templates, ADL failures, Name collisions
- Stretch goal:
    - Exploration of bump allocator support in recovery paths
    - Enhanced crash resilience in interactive workflows
    - Improved value printing in clang-repl

Implementation Plan

We found some cases during testing and analyzed them below. As we continue testing, we expect to find more error recovery issues and work on fixing them step by step.

Case 1: Incomplete Undo of Template Specializations
A limitation in clang-repl is observed when undoing explicit template specializations:

```
clang-repl> template <class T> struct S2 {};
clang-repl> template <> struct S2<int> {};
clang-repl> template <> struct S2<int> { int val =
32; };
// error: redefinition of 'S2<int>'
clang-repl> %undo
clang-repl> template <> struct S2<int> { int val =
32; };
// still reports redefinition
```

Cause
The cleanup logic in IncrementalParser::CleanUpPTU removes declarations from: lookup structures (DeclContext, IdResolver)
but it does not clean up deeper template-related state. In particular:

- The specialization is still stored in ClassTemplateDecl::Specializations

- Its canonical declaration still exists in the AST

When a new specialization is written, Sema directly checks the stored specializations (using findSpecialization). As a result, it finds the old specialization and reports a redefinition.

Impact on clang-repl

- %undo does not fully undo previous input
- Old state leaks into later inputs
- Valid redefinitions fail after rollback

Possible Approach

Extend cleanup to remove or invalidate those specializations. In particular, ensure that entries in ClassTemplateDecl::Specializations are cleared and that any related declarations are no longer reachable through the AST.

Case 2: Parser Recovery Failure on Incomplete Function

In clang-repl, incomplete input can lead to poor parser recovery. For example:

```
clang-repl> void foo() {
```

Behavior

- The compiler repeatedly emits:

error: expected expression

- Errors continue until the limit is reached:

fatal error: too many errors emitted

- The REPL does not recover cleanly after this input.

In Cling:

root [3] void foo() {

root (cont'ed, cancel with .@) [4]

Cling detects that the input is incomplete (e.g., { not closed) and waits for more input instead of parsing immediately.

Cause

After seeing {, the parser enters the function body and runs this loop:

```
while (... Tok.isNot(tok::r_brace) && Tok.isNot(tok::eof)) {
    …
    R = ParseStatementOrDeclaration(...);
}
```

With incomplete input:

- No valid statements are parsed
- No closing } is found

The parser keeps calling ParseStatementOrDeclaration, but makes no progress.

Since the token does not change and no recovery happens, the loop continues and emits the same error repeatedly.

There is no condition to break out when parsing fails without consuming tokens.

Impact

- Repeated error messages
- Poor error output
- Affects parsing of later inputs.

Possible Approach

- Detects situations where the parser is not making progress inside the parsing loop. In interactive mode, there is no explicit EOF token, so the parser may keep running while waiting for tokens like }. This needs special handling when the input is incomplete, especially for function definitions.
- Stop or recover when no tokens are being consumed, to avoid infinite loops or unstable states.

- Optionally delay parsing at the REPL level when the input is clearly incomplete, and wait for more input before continuing.

Case 3: Invalid Specialization Persisting After Failed Definition

This case shows how invalid declarations can persist and affect later inputs in clang-repl.

Scenario

```
clang-repl> template <class T> struct S2 {};
clang-repl> template <> struct S2<int> { int val =;
};
// error: expected expression
clang-repl> template <> struct S2<int> { int val =
32; };
// error: redefinition of 'S2<int>'
```

Behavior

Even though the first specialization has an error, it is still treated as a valid previous definition. Because of this, the second (correct) definition fails with a redefinition error.
In contrast, Cling allows the second definition:

```
root [0] template <class T> struct S2 {};
root [1] template <> struct S2<int> { int val =; };
// error
root [2] template <> struct S2<int> { int val = 32;
};
// succeeds
```

Cause

The specialization is created and stored before the error in the initializer is detected. Even if it is later marked as invalid, it is still:

- Stored in ClassTemplateDecl::Specializations
- Considered during redeclaration checks as an existing definition

Because of this, later definitions see it as already defined and report a redefinition error.

Impact on clang-repl

- Invalid code affects later inputs
- Errors cannot be corrected incrementally
- Behavior differs from other interactive systems

Possible Approaches for REPL

- Investigate how invalid declarations are handled during redeclaration checks
- Consider ignoring or filtering isInvalidDecl() specializations (e.g., via a custom consumer or rollback mechanism)
- Or delay or separate the registration of specializations in REPL mode until they are confirmed to be valid

Automatic Regression Detection

We will use tests that simulate failure and recovery in clang-repl to ensure the system returns to a clean state.

For example:

x.h:

```
#include "test.h"
error_here:
```

Then in clang-repl:

```
clang-repl> #include "x.h"     // triggers error and
recovery
clang-repl> #include "test.h"  // should work
without issues
```

This test checks that all changes from the failed input are properly undone.

We will extend this idea to cover more cases such as:

- template specializations (like the issue described above)
- namespaces
- ADL (Argument-Dependent Lookup)
- macros
- repeated include/undo cycles

This helps automatically detect regressions where leftover state from previous errors affects future inputs.

Timeline

Phase 0: Study and Testing

- Study existing tests and known issues
- Run Clang/Cling tests on clang-repl
- Compare behavior: Cling vs clang-repl
- Identify expected vs actual behavior

Deliverable: Initial set of test cases

Phase 1: Design and Planning

- For each issue:
    - Understand Cling behavior
    - Identify possible changes in Clang (for incremental mode)
- Decide what should be handled in Clang vs clang-repl

Deliverable: Design for fixes

Phase 2: Implementation

- Implement fixes iteratively (from previous phase): Pick issue -> design -> implement -> test
- Focus on: Parser recovery, Sema issues, Template handling

Deliverable: Fixes with tests

Phase 3: Refinement Based on Review

- Improve and stabilize implementations
- Fix remaining issues

Deliverable: Stable behavior across test cases

Phase 4: Testing & Coverage

Add more tests and keep running them to catch issues and avoid regressions. Check behavior between Cling and clang-repl.

Deliverable: test coverage with stable results

Phase 5: Documentation

Write clear notes on design, implementation.

Deliverable: Simple and complete documentation

About Me

I have been working with the LLVM compiler infrastructure and have contributed several patches to the project. This helped me understand the codebase and how to make changes to it.

I also worked on clang-repl during Google Summer of Code, where I explored Clang's incremental compilation. I'm also somewhat familiar with Cling and how its behavior compares with clang-repl.

You can find my contributions here:

https://github.com/llvm/llvm-project/pulls/SahilPatidar

I mainly work with C/C++. I have experience with LLDB and basic knowledge of LLVM ORC JIT and LLVM IR. I also use Git for development.